DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	bianchi-mariano@hotmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El primer problema de este trabajo práctico consistió en dar un algoritmo que sea capaz de decir, dada una secuencia de números, la cantidad de números mínima a eliminar de la misma para que cumpla con cierta propiedad. En este caso, la propiedad que debía cumplir la secuencia modificada era que la misma sea unimodal. Una secuencia es unimodal si la misma cumple que es creciente desde el primer elemento hasta cierta posición y desde dicha posición es decreciente hasta el final[1].

Además, se requirió que la complejidad del algoritmo que resolviera el problema antes mencionado, debía ser estrictamente menor que $O(n^3)$ (donde n es el tamaño de la secuencia).

Se pensaron varias formas de encarar y resolver este problema, finalmente se aplicó la técnica de *programación dinámica*. En las secciones siguientes, daremos una explicación detallada de cómo se implementó el algoritmo, cuál es su complejidad temporal y cómo se comportó frente a distintos valores y tamaños de entrada.

1.2. Explicación

Desde el comienzo resultó bastante difícil encontrar dónde se podía aplicar el *principio de optimalidad* para resolver este problema utilizando *programación dinámica*. Luego, se pudo intuir que este problema, podía ser visto como una combinación de otro problema similar: encontrar la subsecuencia creciente/decreciente más larga. Básicamente, el razonamiento fue el siguiente:

Si para cada posición i de la secuencia sabemos:

- Longitud de la subsecuencia creciente más larga desde el principio hasta i (que contenga al elemento i-ésimo)
- Longitud de la subsecuencia decreciente más larga desde i hasta el final (que contenga al elemento i-ésimo)

entonces podemos decir cuál es el largo de una secuencia unimodal que tiene como "pivote" u objeto más grande al i-ésimo de la secuencia. Finalmente, sólo bastaría saber cuál es el "pivote" que maximice esa longitud, ya que conocer la longitud de la subsecuencia unimodal más larga es equivalente a encontrar la cantidad mínima de elementos a eliminar para transformar la secuencia dada en una secuencia unimodal, que es el problema que necesitabamos resolver.

Entonces, veamos como conseguimos la subsecuencia creciente más larga desde el principio hasta el pivote:

Para construir un algoritmo de programación dinámica definimos:

[1] Debe ser estrictamente creciente/decreciente

- $ascenso_i$ = longitud de la secuencia creciente más larga que termina con el número v_i
- Relación de recurrencia:
  * $ascenso_0 = 0$
  * $ascenso_i = max_{j<i}\{ascenso_j + 1\}$ (con $v_j < v_i$)
- la solucion final: $max_{1 \leq i \leq n}\{ascenso_i\}$

Veamos el pseudocódigo de completar_ascensos para entender el mecanismo:

void **completar_ascensos**(ascenso: vector¡nat¿, v: vector¡int¿)
Complejidad: *O(tam(v)²)*
for $i = 0; i <= tam(v); i++$ **do**
 | ascenso[i] ← maximoAsc(ascenso,v,i) + 1;
end

Donde ascenso es un vector donde se va guardando , en la posicion i, la longitud del mayor ascenso hasta i. La función maximoAsc(ascenso,v,i) indica, dentro del vector ascenso, la maxima longitud de un ascenso hasta la posicion i-1 del vector tal que cumpla la siguiente propiedad:

Si j es el indice del ascenso donde se encuentra el resultado, tiene que ocurrir que: [2]

$$(\forall k \in [0...i), v[k] < v[i])\ ascenso[j] > ascenso[k]$$

Por lo tanto, de esta manera podemos obtener para cada pivote, la longitud del mayor ascenso hasta esa posición.

Veamos un *ejemplo*:
$v = \{9, 5, 2, 8, 7, 3, 1, 6, 4\}$
Las subsecuencias ascendentes más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3

Aquí podemos observar que en cada posición de la fila longitud, se encuentra la longitud de la secuencia creciente más larga hasta ese punto.

[2] ver pseudocódigo en el anexo

Una vez realizado este proceso, podemos obtener, ejecutando un algoritmo equivalente (pero desde el final hacia el principio) que obtenga, para cada posición, la longitud de la secuencia decreciente más larga desde la posición hacia el final. Al ejecutar esté proceso sobre el ejemplo obtendríamos:

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1

Luego, para decidir cual es el mejor pivot, nos basamos en dos filas extra implicitas que podemos ver a continuación :

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1
suma	5	3	2	5	4	3	1	4	3
tam(S) - suma;	4	6	7	4	5	6	8	5	6

Donde $suma_i = ascendente_i + descendente_i - 1 \; \forall i \in [1..tam(s)]$
(esto quiere decir, la suma de las longitudes de la secuencia de mayor longitud creciente hasta i + la longitud de la secuencia de mayor longitud decreciente desde i). Se le resta 1 porque ambas secuencias incluyen al elemento de la posición i.

La fila tam(S)-suma representa, cuantos elementos NO fueron usados para esta "escalera".

Por lo tanto se busca el máximo valor de la fila suma, lo que es equivalente a buscar el mínimo en la fila tam(s)-suma que representa la cantidad minima de borrados a realizar para lograr formar la secuencia unimodal y ese es el resultado devuelto por nuestra función.

1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo implementado en la función *escalerar*, que es el que resuelve el problema que se planteo. Primero veamos el pseudocódigo de la función completa: void **escalerar**(v: vector⟨int⟩)

Complejidad: $O(tam(v)^2)$

```
var max: nat;
var ascenso: vector⟨nat⟩(tam(v));
var descenso: vector⟨nat⟩(tam(v));

completar_ascensos(ascenso,v);
completar_descensos(descenso,v);

for i = 0; i ≤ tam(v); i++ do
    if m then a; ascenso[i] + descenso[i] ≥ max
    x ← ascenso[i] + descenso[i];
end

retornar tam(v) − max + 1;
```

1.4. Anexo: Pseudocódigos

Pseudocódigo de la función descenso y maximoAsc:

void **completar_descensos**(descenso: vector⟨nat⟩, v: vector⟨int⟩)
Complejidad: *O(tam(v)²)*

for $i = tam(v); i \geq 0; i--$ **do**
| descenso[i] ← maximoDes(descenso,v,i) + 1;
end

nat **maximoAsc**(ascenso: vector⟨nat⟩, v: vector⟨int⟩,i: nat)
Complejidad: *O(i)*

var res: nat res ← 0

for $k = 0; k < i; k++$ **do**
| **if** *v[j] < v[i] and res < ascenso[j]* **then** res ← ascenso[j]
end

2. Ejercicio 2

2.1. Introducción

2.2. Explicación

Primero, algunas definiciones:

Def$_1$:
Un grafo G es fuertemente orientable si existe una asignación de direcciones a los ejes del conjunto de ejes del grafo G tal que el digrafo resultante es fuertemente conexo.

Def$_2$:
un puente, **arista de corte** o istmo es una arista que al eliminarse de un grafo incrementa el número de componentes conexos de éste. Equivalentemente, una arista es un puente si y sólo si no está contenido en ningún ciclo.

Teorema [Robbins, 1939] :
Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes (Demostración en el anexo).

Con esté teorema podemos ver que si encontramos al menos 1 puente en nuestro grafo, significa que no podremos orientarlo como queremos, y de lo contrario, si encontramos que no hay ningun puente, podremos orientarlo. Por lo tanto, el algoritmo utilizado realiza exactamente esa comprobación. Veamos como trabaja:

comprobación(Grafo G)

```
1 Complejidad: O(n^3)
2 var eje : int ← 0
3 var n : int ← cantNodos(G)
4 while eje ≤ m do
5 |   k ← RecorridoSinEje(eje,G)
6 |   if k ≠ n then return no se puede
7 |   eje ← eje + 1
8 end
9 return fuertemente conexo
```

RecorridoSinEje(eje,G) es una funcion que recorre el grafo G (con BFS o DFS) sin utilizar la arista eje y retorna la cantidad de nodos visitados, k. Como la forma de recorrer utilizada, solo recorre nodos conectados a la raiz (es decir, al nodo donde comienza el recorrido), quiere decir que el resultado k va a ser n (la cantidad total de nodos de G) si G es conexo. De lo

contrario, si k es menor que n, estamos en presencia de 2 componentes conexas (o más en el caso de $eje = 0$). Por lo tanto, el eje sacado, era un puente.

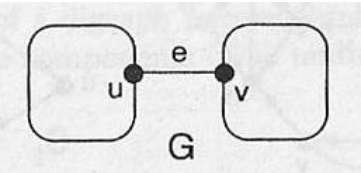
Aclaración:
Cuando eje = 0, representa, recorrer a G completo con todos sus nodos. En este punto podria pasar que G no sea conexo y esta función devolvería el resultado correspondiente.

2.3. Anexo: Demostraciones

Teorema [Robbins, 1939]:
Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes.

Demostración: $\rightarrow$)

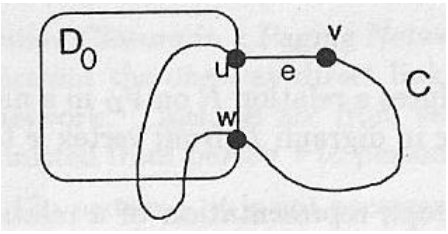
Utilizando el contrareciproco, supongamos que el grafo G tiene una arista de corte e que une a los vertices u y v. Entonces el unico camino entre u y v o v y u en el grafo G es e (ver figura). Por lo tanto para cualquier asignacion de direcciones, el nodo cola(e) nunca va a poder ser alcanzada por el nodo cabeza(e) (ver cuadro 1).



Cuadro 1:

Demostración: $\leftarrow$)

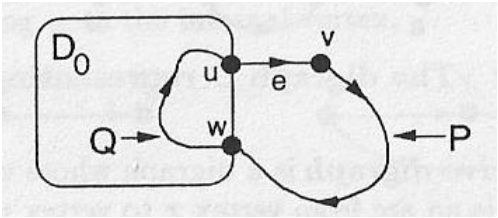
Supongamos que G es un nodo conexo sin aristas de corte. Por esto toda arista en G cae en un circuito de G. Para hacer que G sea fuertemente orientable, empezaremos con cualquier circuito (D_0) de G y dirigiremos sus ejes en una dirección (obteniendo un circuito dirigido). Si el ciclo D_0 contiene todos los nodos de G, entonces la orientación esta completa (ya que D_0 es fuertemente conexo). De lo contrario, hay que elegir cualquier eje e uniendo a un vertice u en D_0 y a un vertice v en $V_G - V_{D_0}$ (ese eje existe ya que G es conexo). Sea C = $\langle u, e, v, ..., u \rangle$ un ciclo que contiene al eje e, y sea w el primer vertice luego de v en C que cae en el ciclo D_0 (ver cuadro 2).



Cuadro 2:

A continuación, direccionamos los ejes de este camino entre v a w, obteniendo el camino dirigido $v-w$ que llamaremos P. Luego direccionamos el eje e desde u hasta v y consideramos el digrafo D_1 que se forma agregando el eje dirigido e a D_0 y todos los vertices y ejes dirigidos del camino P. Como D_0 es fuertemente conexo, entonces hay un camino dirigido de w a u (Q) en D_0 (ver cuadro 3). La concatenación de P con Q y el eje dirigido e forman un circuito simple direccionado que contiene u y los nuevos vertices de D_1. (si los vertices u y w son el mismo, entonces P satiface ser un circuito simple dirigido)

Por lo tanto, el vertice u y todos estos nuevos vetices son mutuamente alcanzables en D_1 y además u y cada vectice del digrafo D_0 tambien son mutuamente alcanzables, y, por lo tanto , el digrafo D_1 es fuertemente conexo. Este proceso puede continuar hasta que el digrafo D_1 para algun $l \geq 1$, contenga todos los vertices de G. En este punto, cualquier asignacion de direcciones hacia las aristas sin direccion restantes completaran la orientación de G, puesto que contendrá el digrafo fuertemente conexo D_l como subdigrafo.



Cuadro 3:

3. Ejercicio 3

3.1. Introducción

El tercer y último ejercicio de este trabajo práctico, consistía en dado un modelo de una prisión decidir si una persona condenada podía escapar o no de la misma. Este modelo fue dado de tal manera que se pudiera representar utilizando un grafo. La idea general del modelo de esta prisión, fue que en la misma haya habitaciones y pasillos. Cada pasillo conectaba dos habitaciones y no había más de un pasillo conectando dos habitaciones. Además, cada habitación podía estar vacía, tener una puerta o tener una llave. Para pasar por una habitación que tenía una puerta había que tener su respectiva llave.

Como requerimiento para este problema, se especificó que su solución debía tener una complejidad estrictamente menor que $O(n^3)$, con n igual a la cantidad de habitaciones.

3.2. Explicación

En primer instancia, se pensó en alguna forma de recorrer el grafo de manera que la misma sirviera para solucionar el problema planteado. Inmediatamente, surgió la idea de utilizar el método BFS para dicho fin.

Seguidamente, se realizaron algunos ejemplos en papel, para ver de que forma se iba a comportar el método elegido para este modelo en cuestión, ya que se no podía utilizar un BFS normal por las restricciones con las que se contaba. Luego de observar este comportamiento, se encontró una forma de resolver el problema utilizando BFS, la cuál se explica a continuación.

La idea utilizada para este ejercicio resultó bastante simple. La misma consiste en los siguientes pasos:

- Se comienza a realizar BFS desde la primer habitación.
- Cada vez que se encuentra una habitación con una llave, ésta se guarda en un conjunto de llaves.
- Cada vez que se encuentra una habitación con puerta pueden suceder 2 cosas:
  - Si ya se había encontrado su llave, se la recorre como si no tuviese puerta.
  - Si no se había encontrado su llave, se la ingresa a una cola donde se alojaran las habitaciones con estas características, sin aplicar BFS en ella.
- Cuando el recorrido BFS termina, puede darse por dos motivos:
  - Que se hayan recorrido todas las habitaciones.
  - Que queden habitaciones sin recorrer en la cola de aquellas que tienen puerta y no se encontró su llave.

Una vez terminado el primer BFS, se verifica si ya se recorrieron todas las habitaciones o, caso contrario, si quedan habitaciones que se puedan visitar[3]. Si sucede lo segundo, entonces se repiten todos los pasos antes mencionados tomando como primer habitación para recorrer con BFS alguna de las que se habían encolado por no tener la llave necesaria para abrirla. Si sucede que se recorrieron todas las habitaciones o que no hay ninguna llave disponible que abra las habitaciones encoladas, entonces se sale de la recursión y se verifica si se visitó o no la última habitación (o habitación de salida). Si se visitó entonces significa que hay forma de salir de la prisión y sino, significa lo contrario.

A continuación se adjunta, a modo de pseudocódigo, los algoritmos utilizados para resolver el problema descrito anteriormente.

bool resolver(const carcel& c)

[3]esto es equivalente a ver si de todas las habitaciones que quedaron encoladas por tener una puerta a la cuál no se tenía acceso, ahora hay alguna de las que sí se tenga llave ahora

```
1  Complejidad: O(n^3)
2  set< int > habitacionesYaVisitadas
3  queue< int > habitacionesLimites
4  set< int > llavesEncontradas
5  int proximaHabitacion
6  bool puedoSeguir = true int contador bool termine = false
7  proximaHabitacion = 0
8  while (puedoSeguir ∧ ¬termine) do
9      recorrerPorBFS(proximaHabitacion, habitacionesLimites,
           habitacionesYaVisitadas, llavesEncontradas, c) // O(n^2)
10     termine = habitacionesLimites.empty();
11     if (¬termine)// O(1)
12     then
13         contador = habitacionesLimites.size()// O(1)
14         while (puedoSeguir &&
               (llavesEncontradas.count(habitacionesLimites.front()) ==
               0))// O(n.log(n))
15         do
16             habitacionesLimites.push(habitacionesLimites.front())
               habitacionesLimites.pop() contador– if(contador == 0) puedoSeguir =
               false
17         end
18         proximaHabitacion = habitacionesLimites.front() habitacionesLimites.pop()
19     end
20 end
21 if (habitacionesYaVisitadas.count(c.cantHabitaciones-1) == 1)// O(log(n))
22 then
23     return true
24 else
25     return false
26 end
```

3.3. Análisis de la complejidad del algoritmo

3.4. Detalles de implementación

3.5. Resultados

3.6. Debate

3.7. Conclusiones